Feb 13th

Thursday, February 13, 2014 10:04 AM

What happen when a semaphore goes below 0?

We get a block, and have to wait for something to increment it.

There are certain litnus tests that the semaphore's have to pass to be valid

- Sync prmitive tests
    - Producer / Consumer
        - Pthread mutex : #include<phtread.h>
            - Type: pthread_mutex_t
            - Init: PTHREAD_MUTEX_INITIALIZER (sets the thread to an unlock state)
                - Pthread_mutex_init(&var, NULL) (this is the static initilization, done for global variables)

In the git repository: there is a program called ProdCons

In this program we are creating a buffer:
It takes a list of numbers
Allocates them via an input stream
This is the producer thread Then fills the buffer
And then starts a consumer thread

We are adding a mutex to make sure that we don't go into dead lock by allowing multiple threads to access the same code

Then we spend some time talking about
Pthread mutex

See Man PTHREAD_MUTEX_DESTROY

If we want to protect something that is great for locks, but if we want to protect something from counting we use semaphore

- Semaphore: #include<semaphore.h>
  Type: sem_t
  Init: sem_init(&semt, shared, value)
  Lock: sem_wait
  Unlock: sem_post

  In the program we deleted the count variable from the structure:
  The semaphore will now regulate the counting by determining if we are waiting or not.
  See sem_wait code and sem_post code

There are certain litnus tests that the semaphore's have to pass to be valid

2nd solution is called Reader Writer

- Reader / writer: all from book
  - Changes
  - Mutex lock
  - Read_count
  - Semaphore rw_mutex = 1
- Writer
  - Wait on rw_mutex
    - Crtical
  - Post on rw_mutex
- Reader
  - Lock mutex
  - Read_count
  - If(read_count == 1)
    - Unlock mutex
- Critical
  - Lock mutex
    - Read count --;
    - If(read_count == 0)
      - Post rw_mutex
    - Unlock mutex

3rd solution is called dining philosophers

Implementation
Lock on ch(n)
Lock on ch(n+1)

This can become a race condition

- Solutions:
  - Limit diners to n-1
  - Only allow pickup if both are available
  - Asymmetric solution
    - All odd diners pick up left then right
    - All even diners pick up right then left

Unlock on ch(n)
Unlock on ch(n+1)

Common errors while using semaphores and mutex's

- Do not
  - Unlock then lock
  - Lock then lock

To avoid common errors

- Using Monitors: see book
  - Helps with ordering issues
  - Gives automatic locking and unlocking of a function
  - Built into Java
    - Class foo {
      - Public synchronized bar(){
      - }
    - }